## 2.2 Syntax of ML.

Edinburgh LCF LNCS 78
Gordon, Milner, Wadsworth 1979

We shall use variables to range over the various constructs of ML as follows:

| | | | | |
|---|---|---|---|---|
| id | ranges | over | ML | identifiers |
| sn | " | " | " | section names |
| ce | " | " | " | constant expressions |
| ty | " | " | " | types |
| db | " | " | " | defined type bindings (see 2.4.4) |
| ab | " | " | " | abstract type bindings (see 2.4.5) |
| d | " | " | " | declarations |
| b | " | " | " | bindings |
| v | " | " | " | varstructs (variable structures) |
| e | " | " | " | expressions |
| s | " | " | " | section commands |

Identifiers, section names and constant expressions are described in 2.2.3 below, types and type-bindings are explained in 2.4 and declarations, bindings, varstructs, expressions and section commands are defined by the following BNF-like syntax equations in which:

1. Each variable ranges over constructs as above.

2. The numbers following the various variables are just to distinguish different occurrences - this will be convenient when we describe the semantics in 2.3.

3. "{C}" denotes an optional occurrence of C, and for n>1 "{C1|C2 ... |Cn}" denotes a choice of exactly one of C1,C2 ... ,Cn.

4. The constructs are listed in order of decreasing binding power.

5. "L" or "R" following a construct means that it associates to the left (L) or right (R) when juxtaposed with itself (where this is syntactically admissible).

6. Certain constructs are equivalent to others and this is indicated by "equiv." followed by the equivalent construct.

### 2.2.1 Syntax Equations for ML.

#### Declarations d

| | |
|---|---|
| d ::= let b | ordinary variables |
| \| letref b | assignable variables |
| \| letrec b | recursive functions |
| \| lettype db | defined types |
| \| abstype ab | abstract types |
| \| absrectype ab | recursive abstract types |

#### Bindings b

| | |
|---|---|
| b ::= v=e | simple binding |
| \| id v1 v2 ... vn {:ty} = e | function definition |
| \| b1 and b2 ... and bn | multiple binding |

#### Varstructs v

| | | |
|---|---|---|
| v ::= () | | empty varstruct |
| \| id | | variable |
| \| v:ty | | type constraint |
| \| v1.v2 | R | list cons |
| \| v1,v2 | R | pairing |
| \| [] | | empty list |
| \| [v1;v2 ... ;vn] | | list of n elements |
| \| (v) | | equiv. "v" |

## Expressions e

```
e ::= ce                          constant

    | id                          variable

    | e1 e2                   L   function application

    | e:ty                        type constraint

    | -e                          unary minus
    | e1*e2                   L   multiplication
    | e1/e2                   L   division
    | e1+e2                   L   addition
    | e1-e2                   L   subtraction
    | e1<e2                       less than
    | e1>e2                       greater than

    | e1.e2                   R   list cons
    | e1@e2                   R   list append

    | e1=e2                   L   equality

    | not e                       negation
    | e1&e2                  R   conjunction
    | e1 or e2               R   disjunction

    | e1=>e2|e3              R   equiv. "if e1 then e2 else e3"

    | do e                        evaluate e for side effects

    | e1,e2                  R   pairing

    | v:=e                        assignment

    | fail                        equiv. "failwith `fail`"
    | failwith e                  failure with explicit token
```



```
    | {if e1 {then|loop} e1'           conditional and loop
       if e2 {then|loop} e2'
                  .
                  .
                  .
       if en {then|loop} en'}
      {{else|loop} en''}

    | e {??|!!} e1 e1'            R   failure trap and loop
       {??|!!} e2 e2'
          .
          .
          .
       {??|!!} en en'
       {{?|!|?\id|!\id} en''}

    | e1;e2 ... ;en                   sequencing

    | []                              empty list
    | [e1;e2 ... ;en]                 list of n elements

    | e where b                   R   equiv. "let b in e"
    | e whereref b                R   equiv. "letref b in e"
    | e whererec b                R   equiv. "letrec b in e"
    | e wheretype db                  equiv. "lettype db in e"
    | e whereabstype ab               equiv. "abstype ab in e"
    | e whereabsrectype ab            equiv. "absrectype ab in e"

    | d in e                          local declaration

    | \v1 v2 ... vn. e                abstraction

    | (e)                             equiv. "e"
```

## Section Commands s

```
s ::= begin {sn}                  begin section {sn}

    | end {sn}                    end section {sn}
```

4. The ML prefixes px and infixes ix are given by:

```
px ::= not | - | do

ix ::= * | / | + | - | . | @ | = | < | > | & | or | ,
```

In addition any identifier (and certain single characters) can be made into an infix. Such user-defined infixes bind tighter than "...=>...|..." but weaker than "or".

Except for "&" and "or", each infix ix (or prefix px) has correlated with it a special identifier $ix (or $px) which is bound to the associated function; for example the identifier "$+" is bound to the addition function, and $@ to the list-append function (see Appendix 3 for the meaning of "$"-ed infixes). This is useful for rebinding infixes (or prefixes) or for passing them as arguments; for example "let $+(x,y) = x*y" rebinds + to mean multiplication, and "f $@" applies f to the append function.

To make an identifier (or admissible single character) into an infix one applies the function "mlinfix" to the appropriate token; for example to infix "oo" one would type "mlinfix `oo`;;" to top-level ML, and then "el oo e2" and "$oo(el,e2)" would be synonymous. If the function one wants to infix is curried, then one can use "mlcinfix" instead of "mlinfix"; for example the effect of doing "mlcinfix `oo`;;" is to make "el oo e2" synonymous with "$oo el e2". More on the functions "mlinfix" and "mlcinfix" is given in Appendix 3. (It is probably better to specify the infix status of an identifier in its declaration, and for this to be local to the scope of the declaration; we leave this to a future version of the system.)

## 2.3 Semantics of ML.

The evaluation of all ML constructs takes place in the context of an environment. This specifies what the identifiers in use denote. Identifiers may be bound either to values or to locations. The contents of locations - which must be values - are specified in the store. If an identifier is bound to a location then (and only then) is it assignable. Thus bindings are held in the environment whereas location contents are held in the store.

The evaluation of ML constructs may either succeed or fail; in the case of success:

1. The evaluation of a declaration, d say, changes the bindings in the environment of the identifiers declared in d. If d is at top level, then the scope of the binding is everything in the current section following d. In "d in e" the scope of d is the evaluation of e, and so when this is finished the environment reverts to its original state (see 2.3.1).

2. The evaluation of an expression yields a value - the value of the expression (see 2.3.2).

3. The evaluation of a section command begins or ends a section. Sections delimit the scope of top level declarations (see 2.3.3).

If an assignment is done during an evaluation, then the store will be changed - we shall refer to these changes as side effects of the evaluation.

If the evaluation of a construct fails, then failure is signalled, and a token is passed, to the context which invoked the evaluation. This token is called the failure token, and normally it indicates the cause of the failure. During evaluations failures may be generated either implicitly by certain error conditions, or explicitly by the the construct "failwith e" (which fails with e's value as failure token). For example, the evaluation of the expression "1/0" fails implicitly with failure token `div`, whilst that of "failwith `tok`" fails explicitly with failure token `tok`. We shall say two evaluations fail similarly if they both fail with the same failure token. For example the evaluation of "1/0" and "failwith `div`" fail similarly. Side effects are not undone by failures.

If during the evaluation of a construct a failure is generated, then unless the construct is a failure trap (i.e. an expression built from "?" and/or "!") the evaluation of the construct itself fails similarly. Thus failures propagate up until trapped, or they reach top level. For example, when evaluating "(1/0)+1000": "1/0" is first evaluated, and the failure this generates causes the evaluation of the whole expression (viz. "(1/0)+1000") to fail with `div`. On the other hand, the evaluation of "(1/0)?1000" traps the failure generated by the evaluation of "1/0", and succeeds with value 1000.

7. "(x,y),[(z.w);()]" matches "(1,2),[[3;4;5];[6;7]]" with x,y,z and w corresponding to 1,2,3 and [4;5] respectively.

### 2.3.2 Expressions

If the evaluation of an expression terminates, then either it succeeds with some value, or it fails; in either case assignments performed during the evaluation may cause side effects. If the evaluation succeeds with some value we shall say that value is returned.

We shall describe the evaluation of expressions by considering the various cases, in the order in which they are listed in the syntax equations.

----ce

The appropriate constant value is returned.

----id

The value associated with id is returned. If id is ordinary, then the value returned is the value bound to id in the environment. If id is assignable, then the value returned is the contents of the location to which id is bound.

----el e2

el and e2 are evaluated in that order. The result of applying the value of el (which must be a function) to that of e2 is returned.

----px e

e is evaluated and then the result of applying px to the value of e is returned.

"-e" and "not e" have the obvious meanings; "do e" evaluates e for its side effects and then returns empty.

----el ix e2

"el&e2" is equivalent to "if el then e2 else false" (so sometimes only el need be evaluated to evaluate "el&e2")

"el or e2" is equivalent to "if el then true else e2" (so sometimes only el needs to be evaluated to evaluate "el or e2")

Except when ix is "&" or "or", el and e2 are evaluated in that order, and the result of applying ix to their two values returned.

"el,e2" returns the pair with first component the value of el, and second component the value of e2; the meaning of the other infixes are given in Appendix 3.

----v:=e

Every identifier in v must be assignable and bound to some location in the environment. The effect of the assignment is to update the contents of these locations (in the store) with the values corresponding to the identifiers produced by evaluating the binding "v=e" (see 2.3.1.1). If the evaluation of e fails, then no updating of locations occurs, and the assignment fails similarly. If the matching to v fails, then the assignment fails with `varstruct`. The value of "v:=e" is the value of e.

----failwith e

e is evaluated and then a failure with e's value (which must be a token) is generated.

```
----{if el {then|loop} el'
     if e2 {then|loop} e2'
                .
                .
                .
     if en {then|loop} en'} {{else|loop} e'}
```

el,e2, ... ,en are evaluated in turn until one of them, em say, returns true (each ei (for 0<i<n+1) must return either true or false). When the phrase following em is "then em'" control is passed to em'; however when the phrase is "loop em'", then em' is evaluated for its side effects, and then control is passed back to the beginning of the whole expression again (i.e. to the beginning of "if el ... ").

In the case that all of el,e2 ... ,en return false, and there is a phrase following en', then if this is "else e'" control is passed to e', whilst if it is "loop e'" then e' is evaluated for its side effects and control is then passed back to the beginning of the whole expression again.

In the case that all of el,...,en return false, but no phrase follows en' - i.e. the option "{{else|loop} e'}" is absent - then empty (the unique value of type ".") is returned.

----e {??|!!} e1 e1'
{??|!!} e2 e2'
.
.
.
{??|!!} en en' {{?|!|?\id|!\id} e'}

e is evaluated and if this succeeds its value is returned.

In the case that e fails, with failure token tok say, then each of e1,e2 ... ,en are evaluated in turn until one of them, em say, returns a token list containing tok (each ei (for 0<i<n+1) must return a token list). If "??" immediately precedes em, then control is passed to em'; however if "!!" precedes it, then em' is evaluated and control is passed back to the beginning of the whole expression (i.e. to the beginning of "e {??|!!} ... ").

If none of e1,e2 ... ,en returns a token list containing tok, and nothing follows en' (i.e. the option "{{ ... } e'}" is absent), then the whole expressions fails with tok (i.e. fails similarly to e).

If none of e1,e2 ... ,en produces a token list containing tok, and "? e'" follows en', then control is passed to e'. But if "! e'" follows en', then e' is evaluated, and control is passed back to the beginning of the whole expression.

If "?\id e'" or "!\id e'" follows en', then e' is evaluated in an environment in which id is bound to the failure token tok (i.e. an evaluation equivalent to doing "let id=tok in e'" is done), and then depending on whether it was "?\" or "!\" that occurred, the value of e' is returned or control is passed back to the beginning of the whole expression respectively.

----e1;e2 ... ;en

e1,e2 ... ,en are evaluated in that order, and the value of en is returned.

----[e1;e2 ... ;en]

e1,e2, ... ,en are evaluated in that order and the list of their values returned. [] evaluates to the null list.

----d in e

d is evaluated, and then e is evaluated in the extended environment and its value returned. The declaration d is local to e, so that after the evaluation of e, the former environment is restored.

\v1 v2 ... vn. e

The evaluation of \-expressions always succeeds and yields a function value. The environment in which the evaluation occurs (i.e. in which the function value is created) is called the definition environment.

1. Simple \-expressions: \v. e

   "\v . e" evaluates to that function which when applied to some argument yields the result of evaluating e in the current (i.e. application time) store, and in the environment obtained from the definition environment by binding any identifiers in v to the corresponding components of the argument (see 2.3.1.1).

2. Compound \-expressions: \v1 v2 ... vn. e

   A \-expression with more than one parameter is curried i.e. "\v1 v2 ... vn. e" is exactly equivalent to "\v1.(\v2 ... .\vn. e) ... " whose meaning is given by 1. above.

Thus the free identifiers in a function keep the same binding they had in the definition environment. So if a free identifier is non-assignable in that environment, then its value is fixed to the value it has there. On the other hand, if a free identifier is assignable in the definition environment, then it will be bound to a location; although that binding is fixed, the contents of the location in the store is not, and can be subsequently changed with assignments.

### 2.3.3 Section commands.

Sections delimit the scope of top level declarations and so provide a form of interactive block structure. They may only be begun or ended at top level. Storage used within a section will be reclaimed (by the garbage collector) when the section is ended.

---- begin {sn}

A new section is begun. The current environment and type definitions (both defined and abstract types) are saved so they can be restored when the section is ended. The new section is named sn if sn is present, otherwise it is nameless.

----end {sn}

The most recently begun section with name sn is ended. If no name is mentioned (i.e. "end;;<return>"), then the most recently opened section (named or nameless) is ended. The effect of ending a